Deconfigurable Microprocessor Architectures for Silicon Debug Acceleration

Nikos Foutris Dimitris Gizopoulos
Department of Informatics & Telecommunications
University of Athens, Greece
{nfoutris, dgizop}@di.uoa.gr

Xavier Vera Antonio Gonzalez
Intel Barcelona Research Center
Intel Labs Barcelona, Spain
{xavier.vera, antonio.gonzalez}@intel.com

ABSTRACT

The share of silicon debug in the overall microprocessor chips development cycle is rapidly expanding due to the ever growing design complexity and the limited efficiency of pre-silicon validation methods. Massive application of short random test programs on the prototype microprocessor chips is one of the most effective parts of silicon debug. However, a major bottleneck and source of "noise" in this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behavior adds long delays in the debug flow since each failing random program must be separately examined, although it does not usually bring new debug information. The development of effective techniques that detect dominant modes of failure among random programs and triage them into common categories eliminate redundant debug sessions and significantly boost silicon debug.

We propose the employment of deconfigurable microprocessor architectures along with self-checking random test programs to reduce the redundant debug sessions and make the triage step of silicon debug more efficient. Several hardware components of high performance microprocessor micro-architectures can be deconfigured while keeping the functional completeness of the design. This is the property we exploit in our silicon debug methodology for the triaging of random test programs. We support our methodology by a hardware mechanism dedicated to silicon debug that groups the failing test programs into categories depending on the microprocessor hardware components that need to be deconfigured for a random test program to be correctly executed. Identical deconfiguration sequences for multiple test programs indicate the existence of redundancy among them and group them together. This grouping significantly reduces the number of failing tests that must be debugged afterwards. Detailed evaluation of the method on an x86 microprocessor demonstrates its efficiency in reducing the debug sessions and thus in accelerating silicon debug.

ISCA'13 Tel-Aviv, Israel.

Categories and Subject Descriptors

C.4 [Computer System Organization]: Performance of Systems – *reliability, availability and serviceability*. B.8.1 [Hardware]: Performance and Reliability – *reliability, testing and fault-tolerance*. B.8.2 [Hardware]: Performance and Reliability – *performance analysis and design aids*.

General Terms

Design, Performance, Reliability, Verification

Keywords

Microprocessor Silicon Debug, Validation, Deconfiguration

1. INTRODUCTION

Aggressive technology scaling and extreme chip integration, combined with the compelling requirement to diminish the time-to-market window have rendered modern microprocessors more prone to design bugs than ever [22], [25], [28]. As a result, *silicon debug* – the process of validating and debugging a new microprocessor design on its first silicon prototype chips – has evolved to a crucial, time-consuming, and labor-demanding step in a microprocessor's development flow. Recent trends [12], [23] show that silicon debug spans more than half of the total chip development cycle, while the ratio between the size of the design and debug teams has reached 2:1. Clearly, an efficient silicon debug process that detects and removes the majority of design bugs before volume production can make the difference between the success and the failure of a microprocessor product.

Silicon debug starts with the arrival of the first prototypes and often continues well after a product has gone to volume production. A comprehensive suite of test programs covering many test scenarios are executed on the prototype chips to detect bugs that can be *anything* from logic/functional bugs, electrical or process-related bugs to mask-related manufacturing defects [15]. Subsequently, for each failing test program (one that does not execute correctly due to a bug), separately, a systematic debug phase is performed by the debug engineers to identify the root cause of the failure.

A critical step in silicon debug is *triage*, the process of analyzing failing test programs and grouping them in "buckets" according to their failure mode. An effective triage process makes actual debug (finding the root cause of the failure) much easier for the debug engineers and allows them to focus on dominant failure modes instead of

spending expensive man power on test programs that fail due to the same underlying issue (the same bug).

One of the most effective parts of silicon debug in terms of bugs detection throughput is the one based on the massive execution of short *random test programs* (each consisting of a few thousand instructions) [2], [4], [11], [19], [34]. All major microprocessor manufacturers have built efficient random test generators that produce trillions of test programs aiming to cover all possible test scenarios defined by the design and debug teams together. Despite its maturity and indisputable efficiency [5], randomness is also the inherent weakness of this part of the process. Random test program-based silicon debug results in the generation of many *redundant* test programs that fail due to the *same or similar* bugs [16], [18], [31]. Every failing random test program consumes several hours or days of man and computational power. This debugging "noise" and overhead is expected to get worse in the future with the increasing design complexities. Clearly, the identification of dominant failure modes among the random test programs that can triage them into categories with common failure modes will significantly reduce the number of debug sessions and will therefore speed silicon debug up by several weeks or months.

In this paper, we propose a silicon debug methodology for microprocessors with the major objective to *automatically triage* the failing random test programs of an *overnight run* in *as small as possible number of "buckets"* of failing tests with common failure modes. This triage obviously leads to *less test programs to be debugged*.

Our methodology optimizes the triage process by exploiting the following property of many hardware components of microprocessors: a component can be "*turned off*" or *deconfigured* while the microprocessor remains functionally complete (i.e. processor's baseline functionality is guaranteed despite the absence of the component). Our silicon debug methodology utilizes such deconfigurable microprocessor architectures and supports them by a dedicated hardware mechanism. Random test programs are grouped in categories based on the *set of components that need to be deconfigured* for the test program to be correctly executed. Deconfigured components are pinpointed as potential hosts of bugs and just a few member of each failing test programs category can be debugged for the identification of the failure root cause.

The proposed methodology operates as follows and aims to deliver a minimal set of failing tests groups after an *overnight* random programs campaign without manual intervention.

- A *self-checking* random test program is loaded for execution on the silicon prototype. The outputs of self-checking random test programs [11], [19], [34] do not need to be compared with golden responses (from pre-silicon simulation) but rather generate a pass/fail indication at the end of their execution. This is a key requirement of the proposed methodology that facilitates re-execution of the test program without external intervention during uncontrolled overnight silicon debug runs.
- If the test program fails, a hardware mechanism (*deconfiguration controller*) decides (based on a pre-defined sequence or dynamically) to deconfigure one of the deconfigurable components of the microprocessor.
- The hardware mechanism arranges for the re-execution of the test program.
- If the test program fails again, another deconfigurable component is "turned off" and the test program is re-executed.
- Finally, if the test program is executed correctly (i.e. bug has been "masked" by the sequence of deconfigurations) the set of components that have been deconfigured is used as a label for a "bucket" of failing tests in the triage process. All test programs that eventually execute correctly after the same sequence of deconfigurations are grouped in the same "bucket". Intuitively, *the bug that causes the failure most probably resides within the components that have been deconfigured*.

As a high-level quantitative example of the aim of the proposed methodology, consider an overnight run of several *millions* of random test programs on the prototypes of a new microprocessor chip. In a traditional silicon debug flow, the next morning, the team of debug engineers will have to perform root cause analysis on a large set of *hundreds* or even *thousands* (depending on the stage of silicon debug – early or late) of failing test programs many of which fail due to the very same underlying bug. The proposed methodology groups the failing test programs in categories with the same component deconfiguration sequence and aims to reduce the number of failing tests that will be eventually debugged to just a few tens.

Our experimental results from the application of the methodology on an x86 microprocessor model report a remarkable reduction in the number of failing random test programs "buckets" and thus the necessary debug sessions.

2. MOTIVATION AND BACKGROUND

Redundant random test programs. The wide-spread adoption of random instruction test generation methods by the main industry players proves the importance of this phase of silicon debug. However, the random nature of the test suites results in the generation of multiple test programs that actually detect the same or similar design bugs because they fail due to their existence [16], [18], [31]. While the same debugging information is shared among all these failing test programs, the debug engineers need to analyze each of them separately, wasting valuable human effort as well as other project resources such as compute time in high performance workstations used for design simulation.

Figure 1 outlines the issue of redundant test programs in a flow without (left) and with (right) a triage method. We

assume that during a time interval a prototype chip executes 1 million random test programs. Among them, 1 out of 10.000 fails due to a design bug (failure rate) and 1 out 10 failing test program is redundant to another (redundant test program rate). The silicon debug flow without a triage method results in 100 debug sessions (one for each failing test program), while for the flow with a triage method, which is able to detect all the redundant test programs, the amount of debug session is reduced to only 10 (i.e. 100 failing test programs grouped into 10 categories).

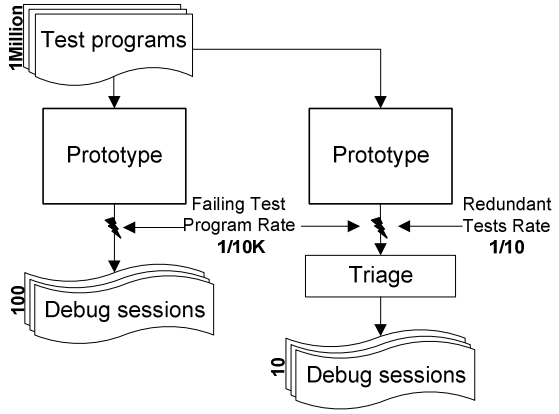

Figure 1: Redundant test program triaging concept.

As our experimental results section shows in detail, we performed a set of experiments to calculate the degree of redundancy among the random test programs in a test suite. Our experimental results show that on average 5 every 1000 failing random test programs detect the same design bug.

Microarchitectural transparency and deconfiguration opportunities. Throughout microprocessors evolution computer architects have devised numerous techniques to improve performance. Superscalar executions paths, multiple functional units, simultaneous multi-threading operating modes, out-of-order execution, dynamic scheduling, branch prediction, data and instruction prefetching are some examples of performance-enhancing mechanisms. All these techniques share a common attribute: microprocessor baseline functionality is guaranteed even without them. For instance, a core will continue to be functionally-complete even when its branch predictor is turned off or a cache memory bank is disabled. Therefore, the integration of these techniques in the hardware of a microprocessor is *transparent* to the instruction set architecture. Because of their sophisticated implementation such components are prone to design bugs. Our methodology aims to triage random test programs that fail due to bugs in such deconfigurable components.

The emergence of the previous techniques in modern microprocessor designs have also contributed to the evolution of Instruction Set Architectures. Sophisticated extensions of the instructions sets have been deployed, targeting to grasp the maximum performance speedup from the enhanced designs. These extensions are essentially built upon a basic set of primitive operations, such as arithmetic and logical operations and memory transactions. For example, the SIMD extension provides the ability to execute multiple arithmetic operations on a data vector, which is essentially comprised of a group of primitive operations. Therefore, equivalent instruction sequences exist [11], which can be used interchangeably to perform the same operation and at the same time these modifications are *transparent* to the semantic correctness of an application.

It is evident that microprocessor architectures built around *deconfigurable components* which can be "turned off", either directly by dedicated hardware at the micro-architecture level or indirectly by software through equivalent instruction sequences as in [11] are feasible. Such deconfigurations will not compromise the functional-completeness of the microprocessor. We employ such architectures in this paper to facilitate the triage step of the silicon debug process.

We studied Intel's Nehalem micro-architecture [13], [14] to estimate the size of logic that is redundant to the baseline functionality of a processor and can be transparently deconfigured through hardware or software.

Components	Size	Instance	Deconfiguration
L1 Instr. Cache	32KB	–	Hardware
L1 Data Cache	32KB	–	Hardware
L2 Cache	256KB	–	Hardware
First level iTLB	128 entries	–	Hardware
Second level iTLB	512 entries	–	Hardware
Instruction Queue	18 entries	–	Hardware
Branch Target Buffer	–	1	Hardware
Conditional Branch Predictor	–	2	Hardware
RAS	16 entries	–	Hardware
Simple Decoder	–	3	Software
Complex Decoder	–	1	Software
MS-ROM	–	1	Software
Instruction Decode Queue	28 entries	–	Hardware
Loop Stream Detector	–	1	Hardware
Micro-fusion	–	1	Hardware
Macro-fusion	–	1	Hardware
Reorder buffer	128 entries	–	Hardware
Reservation Stations	36 entries	–	Hardware
Integer Functional Units	–	9	Either
Floating Point Functional Units	–	3	Either
Load Buffer	48 entries	–	Hardware
Store Buffer	32 entries	–	Hardware

Table 1: Nehalem's deconfigurable components.

Table 1 contains the following information: the first column presents the components of Nehalem core architecture that can be potentially deconfigured. The second column shows the component size (bytes or entries) in cases where part of the component can be deconfigured. The third column presents the number of instances when a single instance of the component must be completely deconfigured. The last column states the way that a component can be deconfigured: through *hardware* at micro-architectural level, through *software* at ISA-level, or *either*. Components with a single instance can be only deconfigured in software when equivalent instruction sequences exist [11].

It is evident from Table 1 that a very large number of microprocessor components, 35 in total, *can be potentially deconfigured* from the design either directly in hardware or indirectly through software or either way. Some storage elements cannot be completely deconfigured. A small part of them must remain enabled to guarantee the baseline functionality of the design. For example, a single entry store buffer, or a 4-entries instruction queue are enough to have a functionally complete design.

In this paper, we focus on the hardware-based deconfiguration of the microprocessor components and the triage support it offers. Previous works [11], [19], [34] have developed self-checking random test programs that can be used for the detection of bugs in microprocessors. Any such method for the development of self-checking random programs can be employed in our methodology.

3. PROPOSED METHODOLOGY

The proposed *deconfigurable microprocessor architecture* can *dynamically* deconfigure several microprocessor modules during the execution of a failing random test program until it is correctly executed: i.e. the bug that causes the failure is masked by the deconfigurations. The main requirements for this to happen during an uncontrolled overnight run of huge numbers of random tests are the following:

Software requirement: the random test programs must be *self-checking* as in [11], [19], [34] so that the failure indication of a test is known to the hardware right at the end of its execution.

Hardware requirement: the *deconfigurable* processor must be equipped with a mechanism which, in case of a failing test program, can: (i) gradually (one at a time) "turn off" its components and (ii) re-execute the failing self-checking test program.

Alternatively, deconfiguration and re-execution can be partly implemented in software (by setting control register values). However, we focus on a hardware implementation because it can collect run-time information from hardware performance counters (existing or new) and it only requires a small part of the microprocessor (our deconfiguration controller explained below) to be bug-free.

The proposed methodology detects design bugs with the following characteristics: (i) their excitation does not depend on the operational conditions (temperature, voltage, frequency) and (ii) they continue to manifest themselves despite the deconfiguration of components from the overall design.

The proposed architecture is outlined in Figure 2 and consists of the following elements.

- *Deconfiguration controller*. This is the main hardware element of the proposed architecture. It interfaces with all the deconfigurable components of the microprocessor (can be several tens of components as shown in the previous subsection) and takes dynamic decisions about the components to be deconfigured during each execution of a random test program.
- *Bypass network*. Controlled by the deconfiguration controller and performs the actual deconfiguration of the hardware components.
- *Profiler*. Each of the deconfigurable components communicates with this module which collects dynamic execution statistics of the random test programs to be considered in the deconfiguration actions. For example, for a memory element the number of write and read operation or for a functional unit the amount of activations can be suitable statistics.

After a self-checking random test fails (mechanism not shown in Figure 2; any approach reported in [11], [19], [34] can be employed), the deconfiguration controller takes the following actions:

- It selects the deconfigurable component that is considered *most susceptible* to contain a bug and turns it off. This decision is based on a *bug susceptibility model* discussed below.
- It arranges for the re-execution of the failing test program; no manual intervention is required.
- If the test program fails again, the deconfiguration controller repeats the previous two steps until the program executes correctly or there are no remaining components to be deconfigured.

The outcome of the operation of the deconfiguration controller for *each self-checking random test program* is a list of components that have been deconfigured. The interpretation of the list provides the following triage-related information.

- *Empty list*. The random test program was correctly executed. No failure detected; no debug action required in the morning.
- *List contains a subset of the deconfigurable components*. The random test program was correctly executed after components $\{C_k, C_n, C_m, C_q\}$ have been deconfigured. The list of components indicates a "bucket" of failing test programs. All test programs ending with the same list of deconfigurations are grouped together. The bug that is the root cause of the failures most likely resides within the deconfigured components. If a deconfigurable component contains

more than one uncorrelated bugs the debug engineers will most probably diagnose them through during root cause analysis using simulation. Even if this does not happen, the execution of the subsequent test programs will detect and pinpoint the buggy component again.

- *List contains all deconfigurable components*. The random test program fails even after all deconfigurable components are turned off. No triage grouping information; the random test must be separately debugged.

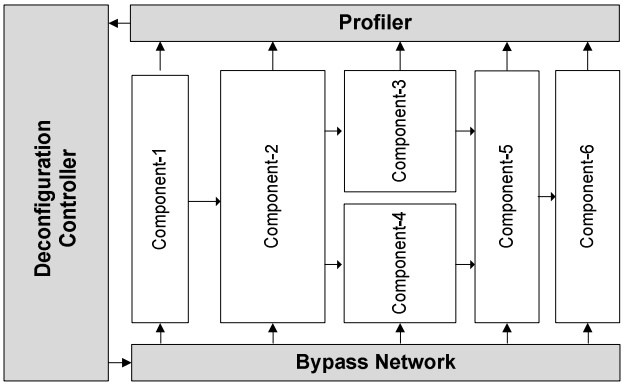

Figure 2: Proposed deconfigurable architecture

The *bug susceptibility model* on which the deconfiguration controller decisions are based is a flexible model that the silicon debug engineers can tune according to the stage of the silicon debug (early or late), the pre-silicon information available and the run-time statistics that can be collected by the profiler.

For each deconfigurable component C_i of the microprocessor, the deconfiguration controller calculates a *bug susceptibility value* S_i (higher value means a more bug-prone component). When a new component must be turned off before a failing test program is re-executed, the component with the highest S_i value that has not been deconfigured yet, is selected and turned off. The information that the deconfiguration controller can use for the calculation of the S_i values of the deconfigurable components is static or dynamic.

Static bug susceptibility information comes from the pre-silicon (simulation-based) debug process. When a microprocessor component C_i is new in a design or if a large number of design bugs have been already found before silicon debug (given that in pre-silicon verification each component has not been exhaustively studied due to the simulation throughput bottleneck, there is a higher probability that more design bugs exist in its design), the debug team can assign it a large bug susceptibility value S_i^{static}. Moreover, static bug susceptibility assignments can be based on the size of the components, on their complexity (larger and more complex components are more bug prone) etc., and the deconfiguration controller can be updated with new values before a new overnight run starts.

Dynamic bug susceptibility information is collected during the execution of random test programs by the profiler component of the proposed architecture. The information may contain activity monitors (from existing or new hardware performance counters or other signals of the design) that show if a component is intensively activated by the particular test program. If this is the case, this is a useful indication that the component is more susceptible to contain the bug that causes the failure of the test program. Therefore, the dynamic bug susceptibility value $S_i^{dynamic}$ of the component C_i must be higher than others with smaller activity during the execution of the failing test program.

In total, the deconfiguration controller calculates an aggregate bug susceptibility value for each component that can be potentially deconfigured:

$$S_i = a * S_i^{static} + (1 - a) * S_i^{dynamic}$$

Parameter a can be tuned (values between 0 and 1) so that decisions lean more towards the static pre-silicon information (large a values) or more towards the dynamic run time statistics (small a values).

The component with the highest susceptibility value S_i is selected to be turned off in the next re-execution of the failing random test program. Bug susceptibility is a metric that can be finely tuned, is based on both dynamic and static information, and bounds the amount of possible test programs re-executions avoiding useless deconfigurations of microprocessor components.

At the end of the repetitive re-executions the failing test is characterized by the set of components that have been deconfigured and have the highest bug susceptibility values. It is therefore, *very likely*, that the failure of the test is due to a bug inside these components. Triaging failing random test programs in "buckets" according to their list of deconfigured components provides useful insight to the debug teams that will start debugging the tests in the morning after an intensive overnight run.

3.1. Cost Implications of the Methodology

Before analyzing the different deconfiguration mechanisms that can be employed in our methodology we discuss the cost implications of the methodology.

Costs related to the deconfiguration infrastructure of the architecture. The components listed in Table 1 (all very common to x86 architectures) can be potentially turned into deconfigurable ones so that the proposed silicon debug methodology is applied. Therefore, the extra hardware adds to the complexity of the design. In the case of some storage elements the need to keep at least some of their entries active while the rest of the component is deconfigured, adds further design modification costs. These hardware modifications come with a positive aspect: the existence of the deconfiguration infrastructure is an added value for the microprocessor because it can be used in the field for the permanent "shut down" of components when they are

diagnosed with hard errors. Employment of deconfiguration (at different granularities) for fault tolerance in the field has been reported in the past [3], [6], [7], [26], [27], [30], [32]. Finally, the cost of deconfiguration depends also on the granularity it is applied. If subsets of entries of a component are separately deconfigured the cost may become high (both for the deconfiguration controller and the bypass network and profiler). We believe that the deconfiguration granularity given in Table 1 (common for x86 microprocessors) is suitable for the needs of silicon debug. If debug engineers are supplied with the information that a short list of 3-4 components have been deconfigured before a test is correctly executed, their debug job is much easier and focuses on the list of these components. In the majority of cases, the list is expected to consist of just one component which is very likely the one with the bug.

Costs related to the dynamic collection of statistics. The profiler component dynamically collects run time statistics when random tests are executed. This feature requires design effort and also increases the area of the microprocessor. If the design team decides to rely on the run time statistics the investment in the design of the profiler and the utilization of its outputs by the deconfiguration controller justifies the cost. However, if the debug team decides to use the deconfiguration controller with a priory known susceptibility values from the pre-silicon effort, the cost of the profiler is saved.

Timing overheads and time savings. The proposed methodology adds a time overhead to the random test phase of silicon debug: each failing random test is repeated one or more times. However, this minimal time overhead is absolutely justified by the large savings in the debugging time of the failing tests. For example, consider an overnight campaign of random test programs executions that is prolonged by a relatively small amount of time (measured in *minutes or an hour* for the entire campaign), required for the re-executions of the failing test programs. Even, assuming a large percentage of failing tests (1%) and an average number of re-executions equal to 5 (i.e. 5 components are deconfigured and a failing test program is repeated 5 times on average) the time overhead for the overnight run will be around 5% (1% times 5 re-executions). In other words, approximately 5% less random test programs will be applied during the night. Even in the worst case, where all available components have to be deconfigured (based on the study we perform on Nehalem's architecture, the total number of deconfigurable components is 35) the extra overhead from the re-executions remains small compared to the expected debug sessions savings. The successful application of the methodology will reduce the number of failing test programs that will need to be debugged from several thousands to just a few tens. This saving is measured in *days or weeks of debug time* and is the major contribution of the method to silicon debug.

3.2. Deconfigurable Architectures

In this section, we review the mechanisms reported in the literature that can be employed to deconfigure the components of a high performance microprocessor. Furthermore, we discuss simple deconfiguration schemes for branch predictors and prefetchers. The deconfiguration granularity can be flexibly tuned and is only limited by the cost implications discussed in the previous subsection.

Previous research [3], [6], [7], [26], [30], [32] proposed various techniques to deconfigure components with permanent faults from a microprocessor design.

The circularly-accessed arrays, the directly-accessed arrays and the functional units comprise the list of processor modules that are often duplicated or contain a high degree of regularity and can be deconfigured. The circularly-accessed arrays, such as the instruction fetch queue, the reorder buffer or the load and store queues, are augmented with a fault map. The fault map communicates with the pointer advancement logic, forcing it to skip an entry that is marked as faulty ("buggy" in silicon debug).

For the cases of functional components and directly-accessed memory arrays, the available deconfiguration solution is to mark the components or memory entries as permanently busy, preventing the microprocessor from issuing further requests to them.

In silicon debug the aforementioned deconfiguration techniques are clearly applicable. However, the deconfiguration granularity could be more coarse-grained (for example, deconfigure half of the reservation stations, or the entire L1 data cache and not parts of it), since localizing a bug at the level of micro-architecture components carries enough information for the debug engineers to root cause the failure.

Modern microprocessors integrate numerous performance-increasing components. Among them, the branch predictors and the data and instruction prefetchers are the most widely used ones. These modules can be deconfigured from the design, since they do not contribute to its functional completeness. Regarding the branch predictors, a simple deconfiguration mechanism can be used, where the component can be bypassed by setting the predictor's state machines permanently to not-taken state. A similar deconfiguration mechanism can be applied to the data and instruction prefetchers. In particular, the prefetcher's queue, where all prefetching requests are buffered, can set permanently the overflow flag. Therefore, all prefetching requests will be dropped and the prefetcher is effectively deconfigured.

3.3. Deconfiguration Controller Design

The main structure of the proposed microprocessor architecture is the deconfiguration controller. The controller can calculate dynamically, at runtime, the susceptibility value of each deconfigurable module and decides the module for the next deconfiguration (i.e.: the module with

the higher probability to be the source of the failure). Figure 3 outlines the structure of the deconfiguration controller in the general case where both static pre-silicon susceptibility information and dynamic run time susceptibility information is utilized.

The deconfiguration controller consists of three memory elements and a combinational part implementing the proposed deconfiguration model. The memory elements are implemented through the allocation of memory-mapped space in the main memory of the prototype. The deconfiguration controller accesses these structures for write/read operations.

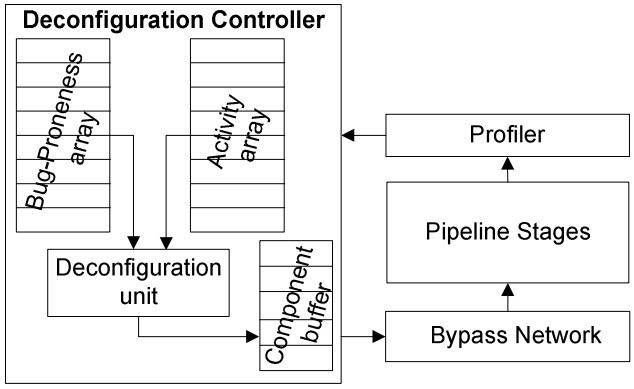

Figure 3: Deconfiguration controller

A *bug-proneness* array is updated with the susceptibility value of each deconfigurable module as assessed from the designer from pre-silicon data. The array is updated at the initialization phase of the silicon prototype chip during test program uploading. A single entry for each deconfigurable component is reserved in the bug-proneness array.

A second array, the *activity array*, is accessed by the profiler and stores the activity (existing or new performance counters and other signals values) of each deconfigurable module during execution of a test program. At the end of test program execution the utilization array is updated with the value of the performance counters. The activity array can contain more than one entry for each deconfigurable component depending on the counters and signals that need to be monitored for the component for a more elaborate decision at run time. Therefore the array may have a size of a few hundreds words.

The deconfiguration unit parses the two memory arrays (bug-proneness and activity) to find the component with the largest value of *bug susceptibility* S_i as described previously. This component is assumed to be the one with the highest probability to be the source of the failure.

The deconfiguration unit output is written to the *component buffer* of the deconfiguration controller. Each entry of this array saves the *id* of the component that will be deconfigured. In every re-execution of the random test program the deconfiguration unit increases the pointer of the component buffer, writes the *id* of the next component to deconfigure and activates the relevant bypass logic.

At the end of the multiple hardware-enabled test program re-executions, the component buffer contents are downloaded along with the remaining memory image of the prototype on the workstation (a server that controls the validation process for a particular prototype) for further analysis in the morning. Before each re-execution of a test program the arrays are reset, since only the components that have not been already deconfigured should be considered in the estimation of the susceptibility model. Figure 4 visualizes the timeline of the operation of the proposed deconfiguration controller.

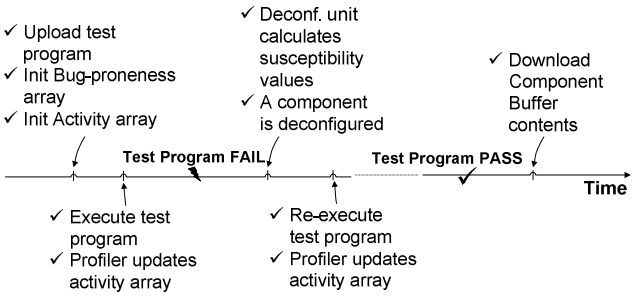

Figure 4: Methodology timeline for each test program.

4. EVALUATION

In this section we evaluate the proposed silicon debug methodology. We first describe the experimental framework. Subsequently, we measure the degree of redundancy among the random test programs generated by our generator (which has been developed following guidelines that major microprocessor companies provide in the literature) for an x86 microprocessor using PTLsim architectural simulator [35]. Finally, using the same framework we demonstrate the benefits of module deconfiguration in random test program triaging for silicon debug acceleration.

4.1. Experimental Framework

For the experimental evaluation of the proposed silicon debug methodology, we set up the tool chain shown in Figure 5 (dashed components are implemented from scratch to evaluate our methodology). The experimental framework consists of the following main modules:

Random Test Program Infrastructure: This module generates random, x86 assembly test programs enhanced with a self-checking capability (i.e. not needing golden responses to compare with in order to conclude about the detection of a design bug). Any of the self-checking methods presented in [11], [19], [34] can be employed; in our case we adopted the method presented in [11] which exploits the diversity property of microprocessor ISAs. The input of the random test program generator is a set with the basic x86 instruction templates. For example, the template of an addition operation is the following: add, <operand$_1$>, [<operand$_2$> | <memory>]. Registers selection, operands value initialization, data memory initialization and instructions sequence are completely randomized. The

output of random test program generator is x86 assembly random test programs of 4K instructions each.

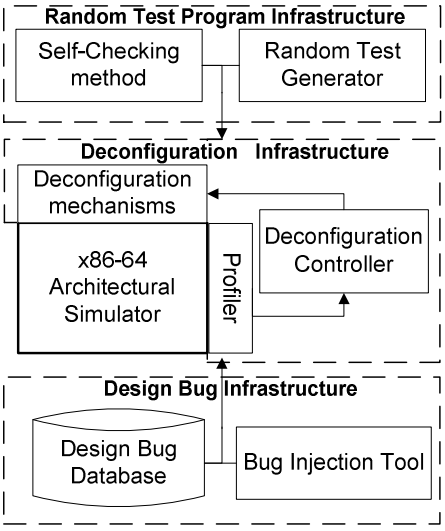

Figure 5: Experimental framework.

Design Bug Infrastructure: The bug injection tool injects design bugs at various locations of an x86 superscalar, out-of-order, single-core design modeled through PTLsim architectural simulator. The bug injection tool interfaces with a design bug database as depicted in Figure 5. The design bug database is populated with a set of logical and electrical bugs that model different design bug conditions in the entire x86 architecture. Specifically, electrical bugs are modeled as transient bit flips at the microprocessor's memory structures as proposed in [20]. Logic bugs are modeled through modification in the architectural simulator's source code, to model their permanent nature [33]. Table 2 summarizes the numbers of logic and electrical bugs injected in the components of the x86 microprocessor model. In total, 1000 design bugs were injected, 500 logical and 500 electrical respectively, covering all pipeline stages and the majority of hardware components of the x86 microprocessor model.

Component	Electrical Bugs	Logic Bugs	Total
Instruction Fetch Queue	50		50
Branch Prediction Unit	50	50	100
Simple Decoder		100	100
Complex Decoder		100	100
Register Renaming	50	50	100
Issue Queues	100		100
Scheduler		100	100
Functional Unit		50	50
Load/Store Queue	100		100
Reorder buffer	150	50	200
Total	**500**	**500**	**1000**

Table 2: Injected design bugs distribution in the x86 microprocessor components.

Deconfiguration Infrastructure: The x86 microprocessor modeled in the PTLsim simulator integrates various structures that can be deconfigured while the processor remains functionally complete. Table 3 lists the deconfigurable components (first column) and the techniques from subsection 3.2 that have been used to implement the deconfiguration on the simulator (second column). The third column presents the initial size of each component, while the last column demonstrates the selected deconfiguration granularity. For example, the size of instruction fetch queue is 32 entries out of which 28 can be deconfigured. The redundant ALU can be also deconfigured.

Component	Deconf. Mechanisms	Initial Size/ Entities	Deconf. Granularity
Instruction Fetch Queue	Fault-map	32 entries/–	28 entries altogether
RAS	Stuck-at	16 entries/–	16 entries altogether
Conditional Predictor	Stuck-at	–/2	2
Instruction Prefetcher	Stuck-at	–/1	1
Data Prefetcher	Stuck-at	–/1	1
BTB	Stuck-at	4K/–	4K entries altogether
Register Renaming Table	Fault-map	16x256	16x128
Issue Queue	Fault-map	16 entries	8 entries altogether
ALU	Busy mode	–/2	1
FPU	Busy mode	–/2	1
Load Queue	Fault-map	48 entries/–	44 entries altogether
Store Queue	Fault-map	32 entries/–	28 entries altogether
Re-order Buffer	Fault-map	128 entries/–	124 entries altogether

Table 3: Deconfigurable microprocessor modules in the x86 model of the PTLsim simulator.

Our deconfiguration infrastructure in the experimental framework integrates a simple profiler component that monitors the activity of the deconfigurable modules and provides the dynamic bug susceptibility data to the deconfiguration controller. We have not implemented all details of the profiler because the analysis we provide in the following subsection does not depend on the type of bug susceptibility that the deconfiguration controller considers (static or dynamic). Future work can analyze the efficiency of different dynamic run time statistics collection by the profiler as well as their exact hardware costs.

4.2. Results

The experimental evaluation of the proposed methodology is divided into two sets of experiments.

First set of experiments. The random test program infrastructure and the design bug infrastructure are used to quantify the degree of redundancy among the random test programs. We need this first set of experiments to support our claim about redundancy which is the main motivation of our work.

For a given set of 1000 design bugs (Table 2) defined in the design bug database and a given set of 10.000 random tests programs generated by the random test program generator, the experimental framework executes each random test program with a single design bug injected at a time and records if the bug is detected or not (test program fails). The graph of Figure 6 shows the number of test programs that fail for each injected bug. The vertical axis shows all the 1000 design bugs injected into PTLsim simulator, while the horizontal axis shows the number of failing random test programs for each bug.

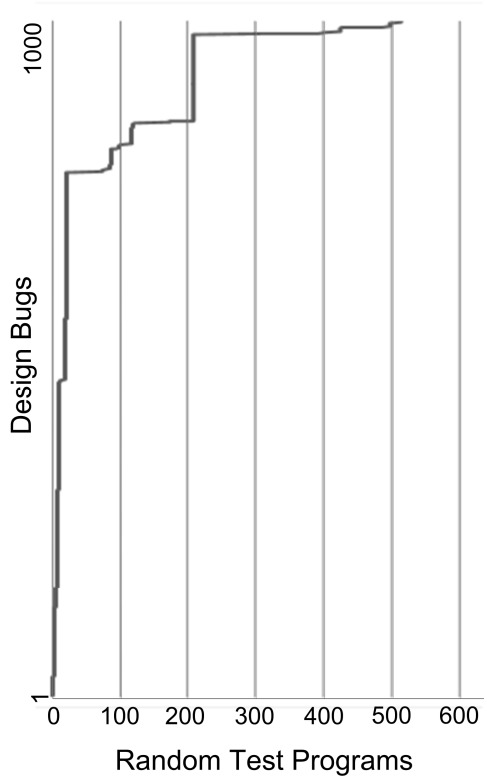

Figure 6: Number of failing random test programs (among 10.000 executed) for each of the 1000 injected design bugs.

The large numbers of redundant test programs are evident in Figure 6. In particular: on average 52 test programs (0.52% of all 10K applied test programs) detect the same design bug (fail due to the bug existence), the maximum number of test programs that fail due to a single bug is 515 (5.15% of all 10K test programs) and the minimum is 2 (0.02% of all 10K test programs). Only 27% (273) of the 1000 injected bugs are detected by more than 30 of the 10K random tests (0.3% of the tests).

Clearly, the motivating observation of this work is valid. If this set of experiments is extrapolated for an overnight run of massive numbers of random test programs, the debug team will have to deal with a very large number of failing tests. Each and every failing test will probably need to be separately debugged a process that may take several days.

Second set of experiments. It aims to demonstrate the benefits of the deconfiguration mechanism for random test program triaging.

Towards this aim, we have selected a set of 10 hard-to-detect logic bugs from the set of 1000 injected bugs (all 10 bugs are detected by a small number of test programs; smaller than the average case) distributed among the deconfigurable modules of PTLsim simulator.

We repeated the experiments only for the subset of the initial 10.000 random test programs that are affected from them (derived from the first set of experiments); these are 341 test programs. A critical difference in this second set of experiments is that *all* 10 *design bugs are together injected from the beginning of the bug injection campaign*, as an attempt to model more accurately the post-silicon validation environment where all bugs can co-exist in the prototype chip. In this set of experiments, of course, the deconfiguration infrastructure shown in Figure 5 is enabled.

Table 4 presents details about the selected design bugs. The first column is the id of each bug, while the second column gives the microprocessor component in which the bug resides. Issue $Queue_1$ and Issue $Queue_2$ refer to different components in the microprocessor design (Issue $Queue_1$ for the integer cluster, and Issue $Queue_2$ for the floating point cluster). The third column shows the number of test programs affected by each design bug when injected individually (from the first set of experiments). For example, the design bugs injected in Issue $Queue_2$ cause 21 of the initial 10.000 test programs to fail. Table 5 describes the 10 bugs.

Bug ID	Microprocessor Component	Failing Test Programs
1	Conditional Predictor	45
2	Return Address Stack	10
3	Issue $Queue_1$	32
4	Issue $Queue_2$	21
5	Floating Point Unit	50
6	Data cache	17
7	Load Queue	47
8	Store Queue	29
9	Reorder Buffer	48
10	Reorder Buffer	42
	Total	**341**

Table 4: Details for the 10 hard-to-detect design bugs.

Microprocessor Component	Bug Description
Conditional Predictor	Update fetch address on branch misprediction fails
Return Address Stack	Incorrect push to stack
Issue $Queue_1$	Dependent uop issued, while producer is waiting in ready-to-writeback state
Issue $Queue_2$	Entry not get flushed on a branch misprediction
Floating Point Unit	Incorrect rounding operation
Data cache	Valid array logic; invalid data read
Load Queue	Load to store aliasing
Store Queue	Store data before address gets valid
Reorder Buffer	Commit entry more than once
Reorder Buffer	Invalid control bit activation

Table 5: Design bugs description.

Figure 7 shows the results from the execution of a subset of random test programs for the set of 10 hard-to-detect design bugs (all 10 bugs injected together – just like in a real prototype chip). In particular, it shows the different "buckets" of failing random test programs that are formed when the proposed methodology is applied (horizontal axis). The vertical axis shows the number of failing test programs of each bucket. In this set of experiments, the deconfiguration sequence is statically determined assuming pre-silicon bug susceptibility data is provided to the deconfiguration controller. The deconfiguration controller deconfigures the microprocessor modules for each pipeline stage starting from instruction fetch. Thus, the sequence of deconfigurations is the following: {Conditional Predictor, RAS, Issue $Queue_1$, Issue $Queue_2$, FPU, Data Cache, Load Queue, Store Queue, ROB}. When all the deconfigurable components from one stage are deconfigured it continues to the next stage. This process is repeated until the random test program is executed correctly or all deconfigurable microprocessor components have been deconfigured.

The application of the proposed methodology, with the deconfiguration mechanisms enabled, results in a triaging of the 341 random test programs in 9 different failure categories shown in Figure 7.

Some observations from this second set of experiments:

- Failure categories 1, 2, 3, 4, 6, 7, and 8 group the test programs that are affected exclusively from the design bugs in the following microprocessor components: Conditional Predictor, RAS, Issue $Queue_1$, Issue $Queue_2$, Data Cache, Load and Store Queues, respectively. As a result, when the deconfiguration controller turned the corresponding microprocessor component off, the bug is "masked" and the test program execution is correct. For example, the design bug in data cache unit, which manifested through the propagation of an incorrect data value, was masked when the data cache block was deconfigured from the design. Furthermore, the design bug in the load queue manifested as an invalid forwarding of loaded data to a dependent instruction. As a result, deconfiguring the load queue entries that hold that buggy information result in a correct execution of the test program.
- Failure category 5 groups 53 random test programs, while the expected number of test programs affected from a design bug in the FPU unit is 50. The reason for that is that these particular test programs (3 from Issue $Queue_2$) were able to detect more than one design bugs (design bugs injected both in the Issue $Queue_2$ and the FPU). As a result, only when *both* buggy microprocessor components were deconfigured the re-execution of the test program results in a correct execution.
- Failure category 9 includes the test programs that fail due to bugs 9 and 10 injected in the Reorder Buffer's logic. The deconfiguration mechanisms were unable to distinguish these design bugs into different categories, since both of them were inside the deconfiguration granularity of the ROB structure. Specifically, these bugs reside in neighboring entries of the re-order buffer and manifest themselves as invalid dependency re-dispatching when a mis-speculation happens. Therefore, the same sequence of deconfiguration results in masking both bugs. This is still very effective because the debug team will certainly focus on the ROB component and it is very likely that it will identify the root cause of both bugs.

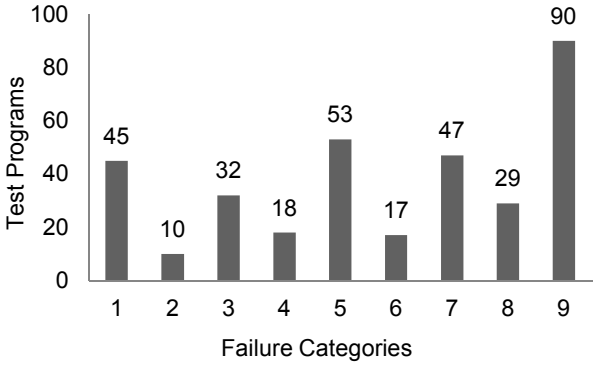

Figure 7: Failure categories for the 341 failing test programs.

In a *traditional silicon debug flow, without the proposed triage mechanism*, the number of failing test in this set of experiments would be 341. This would be exactly the number of debug sessions that the debug team will need to examine starting the next morning. On the contrary, if the *proposed deconfiguration-based silicon debug methodology* is adopted, the number of failing tests remains the same (341) but the number of debug sessions would be *only* 9 (less than 3% of the traditional flow).

Clearly, the proposed flow has a profound impact on the effectiveness of silicon debug and greatly accelerates root

cause analysis by removing the "noise" of redundant random tests that fail due to the same underlying bug. Figure 8 visualizes this reduction in the number of debug sessions when our methodology is applied.

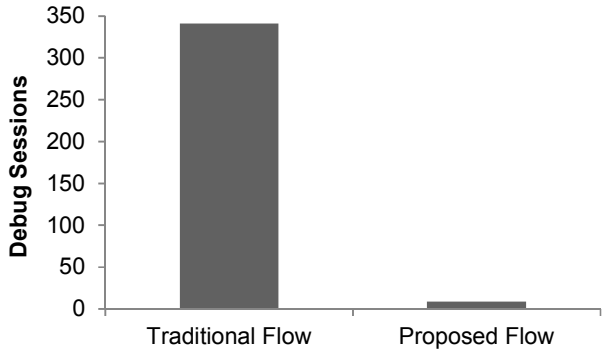

Figure 8: Number of debug sessions (number of failing test programs that must be debugged) in the traditional and the proposed flow.

5. RELATED WORK

There is no work in the literature that reports employment of: (i) deconfigurable microprocessor architectures along with (ii) self-checking random test programs for the optimization of the triage stage of silicon debug.

Triage: In [31] a static grouping of random test programs at generation time through the application of correlation, statistical and pattern recognition analysis methods is proposed. Differently, our methodology dynamically triages the test programs in runtime, based on the bug susceptibility of each component. The work of [11] provides enhanced log information to the debug engineers, facilitating the post-processing analysis of the failing self-checking test programs. On the contrary, the proposed methodology systematically addresses the issue of triaging through the introduction of hardware mechanisms capable to deconfigure a microprocessor design.

Debug [1], [8], [9], [12], [17], [18], [24]: Many proposals introduce various design-for-debug hooks into a design to monitor test execution and extract logging information to facilitate failure analysis. On the contrary, the proposed method acts proactively, in the silicon debug process, reducing the amount of test programs that need to be debugged, by detecting dominant failure modes among the failing random test programs. Furthermore, the massiveness of the silicon debug phase, both in test program execution throughput and in bug detection capabilities, encourages the adoption of high-level debug solutions. The proposed method addresses this challenge, in contrast to the existing research proposals that operate in a very fine granularity. It provides a unified solution for localizing the malfunctioning component throughout the microprocessor design. Obviously, the proposed methodology contributes to the acceleration of the root cause analysis through an improved triaging stage, and complements other silicon debug methods used in the industry.

Fault tolerance [3], [6], [7], [21], [26], [27], [30], [32]: Design deconfiguration is a well-known concept for tolerating hard errors in the field. The proposed methodology employs for first time the concept of deconfiguration in the silicon debug setup and in particular the random test program triaging step; as it is shown throughout the paper, this is not a straightforward task.

6. CONCLUSIONS

Several hardware components of high performance microprocessors can be "turned off" or deconfigured while the functional completeness of the design remains unaffected. In this paper, we combine this property of microprocessor architectures with carefully developed self-checking random test programs to deliver a silicon debug methodology with an optimized triage stage. Redundant failing random test programs during an overnight random test programs execution campaign are grouped in classes each containing test programs that most likely fail due to the same underlying bug. This is decided based on the set of hardware components that need to be deconfigured so that each of the random tests programs is correctly executed.

The main element of the methodology is a hardware deconfiguration controller that prioritizes the deconfigurable components in terms of their bug susceptibility and also arranges the re-execution of random test for as many times as needed to eventually execute them correctly. As a result, in the morning following the overnight run of random test programs, the debug engineers team needs only to focus on the debug of a few failing test programs from each group; a large improvement in the throughput of the debug process.

The proposed methodology has been evaluated in an x86 microprocessor model of a publicly available architectural simulator. Experimental results prove both the validity of the claim that many random test programs fail due to the same bug, and also the large reduction in the debug time that is achieved by the effective triaging of failing tests using the proposed silicon debug methodology.

7. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales "HOLISTIC".

8. REFERENCES

[1] M.Abramovici, P.Bradley, K.Dwarakanath, P.Levin, G.Memmi, D.Miller. "A reconfigurable Design-for-Debug Infrastructure for SoCs", Design Automation Conference (DAC), 2006.

[2] A.Adir, S.Copty, S.Landa, A.Nahir, G.Shurek, A.Ziv, C.Meissner, J.Schumann, "A Unified Methodology for Pre-

Silicon Verification and Post-Silicon Validation", Design, Automation, and Test in Europe Conference (DATE), 2011.

[3] N.Aggarwal, P.Ranganathan, N.Jouppi, J.Smith, "Configurable Isolation: Building high availability systems with commodity multi-core processors", International Symposium on Computer Architecture (ISCA), 2007.

[4] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon and M.Vinov, "Industrial Experience with Test Generation Languages for Processor Verification", Design Automation Conference (DAC), 2004.

[5] T.Bojan, I.Frumkin and R.Mauri, "Intel® First Ever Converged Core Functional Validation Experience: Methodologies, Challenges, Results and Learning", Microprocessor Test and Verification Workshop (MTV), 2007.

[6] F.Bower, D.Sorin and S.Ozev, "A mechanism for online diagnosis of hard faults in microprocessors", International Symposium on Microarchitecture (MICRO), 2005.

[7] F.Bower, P.Shealy, S.Ozev and D.Sorin, "Tolerating hard faults in microprocessor array structures", International Conference on Dependable Systems and Networks (DSN), 2004.

[8] J.Carretero, X.Vera, J.Abella, T.Ramirez, M.Monchiero and A.Gonzalez, "Hardware/Software-based diagnosis of load-store queues using expandable activity logs", International Symposium on High Performance Computer Architecture (HPCA), 2011.

[9] E.Daoud and N.Nicolici, "Embedded Debug Architecture for Bypassing Blocking Bugs During Post-Silicon Validation", *IEEE Transactions on VLSI*, 19(4):559-570, 2011.

[10] D.Feltham, T.Callahan, H.Gartler, L.P.Looi, K.Tiruvallur, R.Mauri, C.Looi, C.Fleckenstein, M.S.Clair, B.Spry, "The Road to Production - debugging and Testing the Nehalem Family of Processors", *Intel Technology Journal*, 14(3):128-146, 2010.

[11] N.Foutris, D.Gizopoulos, M.Psarakis, X.Vera, A.Gonzalez, "Accelerating Microprocessor Silicon Validation by Exposing ISA diversity", International Symposium on Microarchitecture (MICRO), 2011.

[12] Y-C.Hsu, F.Tsai, W.Jong and Y-T Chang, "Visibility Enhancement for Silicon Debug", Design Automation Conference (DAC), 2006.

[13] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation, Nov. 2009.

[14] Intel® 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, Jun.2010.

[15] International Technology Roadmap for Semiconductors, 2009.

[16] D.Josephson, "The Good, the Bad, and the Ugly of Silicon Debug", Design Automation Conference (DAC), 2006.

[17] D.Josephson, "The manic depression of microprocessor debug", International Test Conference (ITC), 2002.

[18] D.Josephson, S.Poehhnan and V.Govan, "Debug Methodology for McKinley Processor", International Test Conference (ITC), 2001.

[19] D.Lin, T.Hong, F.Fallah, N.Hakim and S.Mitra, "Quick Detection of Difficult Bugs for Effective Post-Silicon Validation", Design Automation Conference (DAC), 2012.

[20] R.McLaughlin, S.Venkataraman, C.Lim, "Automated debug of speed path failures using functional tests", VLSI Test Symposium (VTS), 2009.

[21] A.Meixner and D.Sorin, "Detouring: Translating Software to Circumvent Hard Faults in Simple Cores", International Conference on Dependable Systems and Networks (DSN), 2008.

[22] S.Mitra, S.A.Seshia, N.Nicolici, "Post-silicon Validation Opportunities, Challenges and Recent Advances", Design Automation Conference (DAC), 2010.

[23] A.Nahir, "Post-Silicon Validation – Tackling 4 Billions Risks per Second", MEDIAN Workshop, 2012.

[24] S-B.Park, T.Hong and S.Mitra, "Post-Silicon Bug Localization in Processors Using Instruction Footprint Recording and analysis", *IEEE Transactions on CAD*, 28(10):1545-1558, 2009.

[25] P.Patra, "On the Cusp of a Validation Wall", *IEEE Design & Test of Computers*, 27(2):193-196, 2007.

[26] M.Powell, A.Biswas, S.Gupta, S.Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance", International Symposium on Computer Archtecture (ISCA), 2009.

[27] B.Romanescu, D.Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults", International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008.

[28] H.Rotithor, "Post Silicon Validation Methodology for Microprocessors", *IEEE Design & Test of Computers*, 17(4):77-88, 2000.

[29] S.R.Sarangi, B.Greskamp, J.Torrellas, "CARDE: Cycle-Accurate Deterministic Replay for Hardware Debugging", International Conference on Dependable Systems and Networks (DSN), 2006.

[30] P.Shivakumar, S.Keckler, C.Moore, D.Burger, "Exploiting microarchitectural redundancy for defect tolerance", International Conference on Computer Design (ICCD), 2003.

[31] E.Singerman, Y.Abarbanel and S.Baartmans, "Transaction Based Pre-to-Post Silicon Validation", Design Automation Conference (DAC), 2011.

[32] J.Srinivasan, S.Adve, P.Bose, J.Rivers, "Exploiting structural duplication for lifetime reliability enhancement", International Symposium on Compute Architecture (ISCA), 2005.

[33] S.Sudhakrishman, L.Su and J.Renau, "Processor Verification using hwBugHunt", International Symposium on Quality Electronic Design (ISQED), 2008.

[34] I.Wagner and V.Bertacco, "Reversi: Post-Silicon Validation System for Modern Microprocessors", International Conference on Computer Design (ICCD), 2008.

[35] M.Yourst, "PTLsim: A cycle Accurate Full System x86-64 Microarchitectural Simulator", International Symposium on Performance Analysis of Systems and Software (ISPASS), 2007.